LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-TR-664285

Code profiling Using Multiple Profilers on Multiple Machines

K. K. Ghosh

November 13, 2014

Disclaimer

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Code Profiling Using Multiple Profilers on Multiple Machines

Koushik Ghosh

1 Tools Inter-comparison Project (TIP)

1.1 Motivation and Summary

Use of performance analysis tools to understand bottlenecks of large simulation codes is a common practice that is becoming even more significant with the advent of many-core, multi-core and accelerator-enabled HPC systems. While several sophisticated tools are available, each has assorted strengths and weaknesses.

Ease of use is an important factor that varies from tool to tool. In general, the learning curve of a complicated tool will often discourage an eager application developer from investing too much time into learning how to use it.

Tool robustness is a quality that is of utmost importance to a user/developer of a large HPC simulation code. The unavoidable increase in cores per chip, chips per node and nodes per system is also demanding scalability. In addition, there are software issues related to language features such as template-heavy C++ codes and interlanguage issues such python interface to C++ codes. Improving robustness will definitely/always be a work in progress for tool developers. The best way to improve quality and robustness is to stress-test the tools with these million-line production codes and keeping the tool developers updated about the outcome of the use of the tools with real-life data-sets. Dealing with large amount of profile data and/or reducing the amount of data collected without compromising quality of information also is a feature that has to be offered by a robust tool.

Tool portability among various platforms is another issue, especially for open-source tools. Within the DSRC arena there are several large supercomputing systems, such as AMD-based systems from Cray, Intel-based systems from Cray, and Intel-based systems from IBM. The tools should be usable on all of these systems and should provide consistent, trustworthy performance data regardless of the platform.

Finally there is always a request from users for “insight”, instead of mountains of performance data. This issue is much harder to tackle than it seems but is something that could use innovative work, especially as we approach extreme-scale computing.

To address the concerns voiced above, it was thought fit to initiate a tool inter-comparison project that would involve looking at multiple, tools, evaluate them on multiple DSRC machines and execute multiple codes with different computational characteristics.

- Understand tools’ features: complexity, flexibility, ease of use, ramp-up time
- Check if performance data from tools match one another
- Check robustness of tools and investigate techniques for robust and reliable measurements
- Build confidence in tools’ output
- Look for application specific differences exposed by executing multiple simulation codes
- Look for platform dependent differences in our ability to deploy these tools
- Look into strategies for managing size of profiles and reducing volume of post-processing data

2 General Capabilities and Strength of Tools

2.1.1 Sampling, Instrumentation, Tracing

Most tools support two categories of experiments

- Sampling - Build asynchronous experiments
  - Capture values from the program counter and/or call stack at specified intervals or when a specified counter overflows. Later, if requested, display the call path information about the application as well as inclusive and exclusive timing data for functions or code-blocks.
- Sampling can be useful as a starting point
  - provides first overview of the work distribution
- Tracing - Event-based experiments
  - count some events such as MPI, I/O, the number of times a specific system call is executed
- Tracing provides most useful information
  - high overhead if the app runs on a large number of cores for a long period of time

2.1.2 Binary Instrumentation

2.1.3 Source Instrumentation

2.1.4 Hardware Counters and their Values Using PAPI - advantages and disadvantages

What is PAPI?
Open Source software from U. Tennessee, Knoxville: http://icl.cs.utk.edu/papi
Middleware to provide a consistent programming interface for the performance counter hardware found in most major micro-processors.
Countable events are defined in two ways:

- Platform-neutral preset events
- Platform-dependent native events

Presets can be derived from multiple native events.
All events are referenced by name.

2.2 Cray Performance Analysis Tools (PAT)

Using PAT is a three step process:

use pat_build first to instrument your program and capture performance data
use pat_report to process the raw data and convert it to .ap2 format
use Cray Apprentice2, to visualize and explore the resulting data files

Strengths:

- Provide a complete solution - instrumentation, measurement, analysis, visualization of data
- Performance measurement and analysis on large systems
- Automatic Profiling Analysis
- Load Imbalance
- HW counter derived metrics
- Predefined trace groups provide performance statistics for libraries called by program (blas, lapack, pgas runtime, netcdf, hdf5, etc.)
- Observations of inefficient performance
- Data collection and presentation filtering
- Data correlates to user source (line number info, etc.)
- Support MPI, SHMEM, OpenMP, UPC, CAF, OpenACC
- Access to network counters
- Minimal program perturbation

2.3 Open Speedshop (OSS)

Open|SpeedShop

Strengths:

- Comprehensive performance analysis for sequential, multithreaded, and MPI applications
- No need to recompile the user's application.
- Supports both first analysis steps as well as deeper analysis options for performance experts
- Easy to use GUI and fully scriptable through a command line interface and Python
- Supports Linux Systems and Clusters with Intel and AMD processors
- Extensible through new performance analysis plugins ensuring consistent look and feel
- In production use on all major cluster platforms at LANL, LLNL, SNL, NASA and DoD

2.4 TAU

TAU stands for Tuning and Analysis Utilities. TAU is a 18+ year project.

- Comprehensive performance profiling and tracing
- Integrated, scalable, flexible, portable
- Targets all parallel programming/execution paradigms
- Integrated performance toolkit

- Instrumentation, measurement, analysis, visualization
- Widely-ported performance profiling / tracing system
- Performance data management and data mining
- Open source (BSD-style license)
- Easy to integrate in application frameworks
- *http://tau.uoregon.edu*

2.5 Other Tools Considered

2.5.1 MAP (Allinea)

2.5.1.1 Features

- clear, low-overhead MPI profiling
- Re-compilation / instrumentation of code not needed
- Compile code with -g
- start Allinea MAP as you would mpiexec:
    - $ map -n 128 gamess inputName
- shares the common Allinea tools platform developed for Allinea DDT
- tested on everything from the world's largest machines to embedded processors.
- After program finishes
    - MAP shows the lines of source code that took the longest
    - time spent computing in green and communicating in blue
- MAP look and feel similar to DDT's - the source code, the parallel stack view
- Generates reasonable amount of data
- Uses adaptive sampling rates + on-cluster merge technology
    - Controls the amount of data recorded – critical for large core counts
- All metrics turned on, all the time (5% wall-clock stated overhead – not tested at DSRC yet)
- GUI display
    - Shows memory usage, floating-point calculations and MPI usage at a glance
    - CPU view shows the percentage of vectorized SIMD instructions, including AVX extensions used in each part of the code
    - Shows how the amount of time spent in memory operations varies over time and processes
    - cache use efficiency
    - Uses can zoom timeline, isolate a single iteration and explore its behavior in detail
    - Everything shows aggregated data, shows distributions with outlying ranks labelled

2.5.2 HPC Toolkit (Rice University)

2.5.2.1 Features

- An integrated suite of tools for measurement and analysis of program performance
- Serial, threaded (pthreads/OpenMP), MPI, and hybrid (MPI+threads) parallel codes
- Works on computers ranging from multicore desktop systems to the largest supercomputers.
- Uses statistical sampling of timers and hardware performance counters,
- Collects accurate measurements of a program's work, resource consumption, and inefficiency
- Attributes them to the full calling context in which they occur.
- Works with multi-language, fully optimized applications
- Executables can be statically or dynamically linked.
- Uses sampling, measurement has low overhead (1-5%),
- Scales to large parallel systems.
- presentation tools enable (both within and across nodes of a parallel system)

- rapid analysis of a program's execution costs,
- inefficiency
- scaling characteristics

2.5.3 Vampir-trace/scoreP/Vampir (TU Dresden)

2.5.3.1 Features

- Automatic function instrumentation:
  - Using specifics of the underlying compiler
  - Using tools like TAU and Dyninst
- Links instrumentation libraries
  - MPI
  - POSIX Threads
  - libc: I/O, memory, and other system calls
  - CUDA, CUPTI
- Instruments OpenMP statements (using Opari)
- Launches application with an appropriate test-set
- Measurement library performs:
  - Event data collection
  - Triggered by function calls
  - Records selected performance metrics
  - Precise time measurement
  - Parallel timer synchronization
  - Filtering and grouping of function calls
  - Monitor accelerator
- Measurement is controlled via environment variables
- Event records stored in internal buffer
  - Written to file at the very end
- Improved OPARI2 instrumentation for OpenMP

2.5.4 mpiP (LLNL)

2.5.4.1 Features

- mpiP is a lightweight profiling library for MPI applications
- collects statistical information about MPI functions
- low overhead
- generates much less data than tracing tools
- All the information captured by mpiP is task-local
- uses communication during report generation, to merge results from all of the tasks into one output file.
- tested on a variety of C/C++/Fortran applications from 2 to 262144 process
- tested on a 262144-process run on the LLNL Sequoia BG/Q system

2.6 Comparison of Strengths

Tool	Strength	AVUS	HYCOM	GAMESS

		Evaluated / Working?		
Open Speedshop	MPI, HW counters,Sampling, Comparison	Yes	Yes	Yes
Cray PAT	MPI, Binary instrumentation Feedback/Insight	Yes	Yes	Yes
TAU	MPI, HW Counters, Threads, Sampling, Source/Binary Instrumentation	Yes	Yes	Yes
MAP	MPI/FP/vecorization/memory, No PAPI support		Yes	Yes
VAMPIR/scoreP	Communication Matrix		Yes	
mpiP (LLNL/CASC)	Lightweight MPI	Yes	Yes	Yes
HPC Toolkit	MPI, Threads, Scalability			

3 Description of Codes Used

3.1 AVUS

AVUS is an acronym for Air Vehicles Unstructured Flow Solver, formerly known as COBALT60.
AVUS is a software package to provide unstructured grid solution of unsteady, ideal gas, euler/navier-stokes equations. The software used in our project is software version dated: 25 AUGUST 2010.

3.2 HYCOM

HYCOM, from hycom.org, stands for HYBRID COORDINATE OCEAN MODEL.

Traditional vertical coordinate choices [z-level, terrain-following (sigma), isopycnic] are not by themselves optimal everywhere in the ocean. Ideally, an ocean general circulation model (OGCM) should (a) retain its water mass characteristics for centuries (a characteristic of isopycnic coordinates), (b) have high vertical resolution in the surface mixed layer (a characteristic of z-level coordinates) for proper representation of thermodynamical and biochemical processes, (c) maintain sufficient vertical resolution in unstratified or weakly-stratified regions of the ocean, and (d) have high vertical resolution in coastal regions (a characteristic of terrain-following coordinates).

The hybrid coordinate is one that is isopycnal in the open, stratified ocean, but smoothly reverts to a terrain-following coordinate in shallow coastal regions, and to z-level coordinates in the mixed layer and/or unstratified seas.

3.3 GAMESS

GAMESS, which is an acronym for The General Atomic and Molecular Electronic Structure System, is a general ab initio quantum chemistry package (**http://www.msg.ameslab.gov/gamess/index.html**). Briefly, GAMESS can compute SCF wavefunctions ranging from RHF, ROHF, UHF, GVB, and MCSCF. Correlation corrections to these SCF wavefunctions include Configuration Interaction, second order perturbation Theory, and Coupled-Cluster approaches, as well as the Density Functional Theory approximation. Excited states can be computed by CI, EOM, or TD-DFT procedures. Nuclear gradients are available, for automatic geometry optimization, transition state searches, or reaction path following. Computation of the energy hessian permits prediction of vibrational frequencies, with IR or Raman intensities. Solvent effects may be modeled by the discrete Effective Fragment potentials, or continuum models such as the Polarizable Continuum Model. Numerous relativistic computations are available, including infinite order two component scalar corrections, with various spin-orbit coupling options. The Fragment Molecular Orbital method permits use of many of these sophisticated treatments to be used on very large systems, by dividing the computation into small fragments.

4 Description of Supercomputing Platforms Used for the Study

4.1 Cray XE6 AMD Opteron 6300 (Interlagos) 2.5 GHz @ ERDC

4.1.1 System Location

ERDC DSRC – garnet.erdc.hpc.mil 

4.1.2 Node Description

Cray XE6 Node Configuration

	Login Nodes	Compute Nodes
Total Cores | Nodes	128 | 8	150912 | 4716
Operating System	SLES 11	Cray Linux Environment
Cores/Node	16	32
Core Type	AMD 64-bit Opteron	AMD Interlagos Opteron
Core Speed	2.7 GHz	2.5 GHz
Memory/Node	128 GBytes	64 GBytes
Accessible Memory/Node	8 GBytes	60 GBytes
Memory Model	Shared on node.	Shared on node. Distributed across cluster.
Interconnect Type	Ethernet	Cray Gemini

4.2 SGI ICE X Intel Xeon E5 2670 0 (Sandybridge) 2.6 GHz @ AFRL

4.2.1 System location

AFRL DSRC - spirit.afrl.hpc.mil 

4.2.2 Node Description

SGI ICE X Node Configuration

	Login Nodes	Compute Nodes
Total Cores | Nodes	128 | 8	73440 | 4590
Operating System	RHEL 6	
Cores/Node	16	
Core Type	Intel Xeon Sandy Bridge	
Core Speed	2.6 GHz	
Memory/Node	64 GBytes	32 GBytes
Accessible Memory/Node	62 GBytes	30 GBytes
Memory Model	Shared on node.	Shared on node. Distributed across cluster.
Interconnect Type	FDR 14x Infiniband	FDR 14x Infiniband;Enhanced LX Hypercube

4.3 CRAY XC30 Intel Xeon E5 2697 (Ivybridge) 2.7 GHz @ NAVO

4.3.1 System location

NAVY DSRC - armstrong|shepherd.navo.hpc.mil 

4.3.2 Node Description

Cray XC30 Node Configuration

	Login Nodes	Compute Nodes
Total Cores | Nodes	288 | 12	56880 | 2370
Operating System	SUSE Linux	Cray Linux Environment
Cores/Node	24	
Core Type	Intel Xeon E5-2697v2	Intel Xeon E5-2697v2
Core Speed	2.7 GHz	
Memory/Node	256 GBytes	64 GBytes
Accessible Memory/Node	240 GBytes	63 GBytes

Memory Model	Shared on node.	Shared on node. Distributed across cluster.
Interconnect Type	Ethernet	Cray Aries

5 Building Executables for Profiling

5.1 General Considerations

5.1.1 Static vs. Dynamic Executables

Linux Library Types:
There are two Linux C/C++ library types which can be created:

- Static libraries (e.g libmpich.a):
  - Library of object code which is linked with, and becomes part of the application.
    - Increases the size of the executable
    - when a library needs to be updated you'll need to compile the new library and then recompile the application to take advantage of the new library.
- Dynamically linked shared object libraries (e.g. libmpich.so):
  There is only one form of this library but it can be used in two ways.
  - Dynamically linked at run time but statically aware. The libraries must be available during compile/link phase. The shared objects are not included into the executable component but are tied to the execution. Some advantages are
    - Underlying dynamic library could be updated but the executable does not have to be linked
    - Executable footprint is smaller
  - Dynamically loaded/unloaded and linked during execution (i.e. browser plug-in) using the dynamic linking loader system functions – probably not a concern for users' apps.
- Example from ERDC DSRC
  [kghosh@garnet07:/opt/cray/mpt/6.3.0/gni/mpich2-cray/81/lib]$ ll libmpich\.*
  lrwxrwxrwx 1 bin bin 15 May 21 19:02 libmpich.a -> libmpich_cray.a
  lrwxrwxrwx 1 bin bin 16 May 21 19:02 libmpich.so -> libmpich_cray.so

Cray ftn
-h shared Creates a library which may be dynamically linked at runtime.
  Please use ftn -shared with the generic ftn command on Cray systems
-h static Linker uses the static version of the libraries, not the dynamic version of the
  libraries, to create an executable file.
  Please use ftn -static with the generic ftn command on Cray systems
-h dynamic Compiler driver links dynamic libraries at runtime to create a
  dynamically linked executable files and may
  Do not use with the -h static or -h shared options.
  Please use ftn -dynamic with the generic ftn command on Cray systems

OSS
When shared library support is limited the normal manner of running experiments
in Open|SpeedShop doesn't work. You must link the collectors into the static executable.
Currently Open|SpeedShop has static support on Cray and the Blue Gene P/Q platforms.
You must relink the application with the osslink command to add support for the collectors.
The osslink command is a script that will help with linking.

Calls to it are usually embedded inside an application's makefiles.

The user generally needs to find the target that creates the actual static executable and create a collector target that links in the selected collector. The following is an example for re-linking the smg2000 application.

```
smg2000_pcsamp.x: ${OBJS}
@echo "Linking" $@ "... "
osslink -v -c pcsamp ${CC} -o smg2000_pcsamp.x ${OBJS} ${LFLAGS}
```

There are similar commands for linking an executable for use with experiments pertaining to hwcsamp, usertime

5.1.2 Instrumentation of User Code

Most tools use some form of instrumentation that could be performed during compilation of the code. The compiler inserts hooks that are used at runtime to call tool-specific functions that perform data collection based on users' requests or the tools default data collection policy.

An example would be from TAU inserting function calls such as

```
5ca288:     e8 73 73 fd ff       callq  5a1600 <Tau_lite_start_timer>
5ca2a2:     e8 79 7a fd ff       callq  5a1d20 <Tau_lite_stop_timer>
```

Tools can also take an executable and inject data collection probes (e.g. Cray's pat_build, see description of use of pat_build in Section x.y.z.w). One might see evidence of injecting start/stop functions in the instrumented executable:

```
4530d4:     e8 70 49 fe ff       callq  437a49 <__pat_hwpc_start>
45314d:     e8 64 4d fe ff       callq  437eb6 <__pat_hwpc_stop>
```

(FYI, the utility "objdump –s –D a.out_instrumented" was used to look for these hooks.)
In addition to compiler instrumentation, instrumentation of the executable by a stand alone tool like pat_build, a user can MANUALLY inject start/stop statements using the tool vendor specific API. E.g.
ToolX_START ("My_instrumented_region");
ToolX_STOP ("My_Instrumentated_region");
This aforementioned "source instrumentation" mechanism will require recompilation and relinking. The tool will treat the block of code fenced within START and STOP using the user-chose symbolic name "My_instrumented _code". One can add as many of these as one needs to refine data collection with a function of importance.

5.1.3 Dyninst API

The Paradyn project (UofWisconsin/UofMaryland) develops technology that aids tool and application developers in their pursuit of high-performance, scalable, parallel and distributed software. The primary project, Paradyn, leverages a technique called dynamic instrumentation to efficiently obtain performance profiles of unmodified executables. This dynamic binary instrumentation technology is independently available to researchers and developers via the Dyninst API. Many tools use this technology to aid them in their data collection.

5.1.4 Modules and Dotkits

On DSRC systems users have a wide variety of choices for Programming Environments. The loaded modules have to be compatible with the tools' expectations. Most of the time the default versions will work in unison but when it doess not the tools will try to point to the incompatibility and a solution. Of course CCAC is always a good place to contact if there is an issue.

5.1.5 Compiler versions

The default compilers selected by DSRC's system staff are a good place start. But in case there are unexpected errors, users should be prepared to move to a newer or older of the version. For testing purposes it might also make sense to try a compiler from a different vendor, e.g. choosing between Intel, PGI, Cray or GNU compilers. For a large code, switching compilers is not an easy chore as the newly chosen compiler might be less forgiving than the previous one.

5.1.6 MPI Implementation and Compatibility Issues

Most Cray systems will have cray-mpich/6.3.0 installed. This is fine. Your systems (Cray XC30) might have cray-mpich/7.0.0 (based on MPT 7.0.0) available. The advantages of switching to the cray-mpich/7.0.0 are being investigated. Use of module cray-mpich/5.6.x or cray-mpich2 is not recommended.

5.1.7 Wrappers and scripts for compilation and linking

We are all familiar with the use of wrappers/scripts in one form or another. They export necessary environment variables, hide details of a sequence of steps and prevent errors from occurring from accidental misuse once the wrappers have been tested thoroughly. MPI wrappers such as mpif90, mpicc, mpicxx etc. can encapsulate details related to include directories, libraries needed, etc. to build executables.
Tools will often use wrappers too, in addition to loading a module.
Some examples follow.
TAU uses compiler wrappers such as tau_f90.sh, tau_cc.sh and tau_cxx.sh to correctly setup the compiling and linking command it needs to do its job.
Cray PAT uses a wrapper called pat_build to instrument an executable.
OSS needs a script called osslink to build a static executable.

5.2 OSS Build Procedures

Nothing special is required for building an executable for use with OSS. Compile and link the code with the optimization flags normally used for the app. If a static executable is built then the discussion in Section 5.1.1 is relevant:

```
smg2000_pcsamp.x:       ${OBJS}
osslink -v -c pcsamp ${CC} -o smg2000_pcsamp.x ${OBJS} ${LFLAGS}
```

There are similar commands for linking an executable for use with experiments pertaining to hwcsamp, usertime

5.3 PAT Build Procedures

- pat_build (stand-alone utility) that instruments applications for performance collection
- Requires no source code or makefile modification
  - Automatic instrumentation at group (function) level
  - Groups: mpi, io, heap, math SW, …
- Performs link-time instrumentation
  - Requires object files
  - Instruments optimized code
  - Generates stand-alone instrumented program
  - Preserves original binary
- Create an instrumented executable easily using Automatic Program Analysis

```
module load perftools      # Access performance tools SW
make clean
make                       # Build application, use –h keepfiles to keep intermediate files
pat_build -O apa hycom     # will create instrumented executable hycom+pat
```

5.3.1 The .apa File

Here is atypical .apa file that can be modified and used to recreate a binary that suits better the users data collection need.
The variable PAT_RT_PERFCTR can be reset via the option:
-Drtenv=PAT_RT_PERFCTR=default|something_else

```
#  You can edit this file, if desired, and use it
#  to reinstrument the program for tracing like this:
#          pat_build -O report3.apa
#  These suggested trace options are based on data from:
#    gamess.00.x+pat+23702555-175s.ap2
#       Collect the default HWPC group.
-Drtenv=PAT_RT_PERFCTR=default
#       Libraries to trace.
  -g mpi
#       User-defined functions to trace, sorted by % of samples.
  -u   # Enable tracing of user-defined functions.
# 29.51% 1451  bytes
        -T dirfck_rhf_
# 4.07% 16554  bytes
        -T shellquart_
# 3.91% 39365  bytes
-o gamess.00.x+apa                              # New instrumented program.
```

5.3.2 Keeping Intermediate Files

With Cray compilers (CCE) use –h keepfiles.
With Intel compilers (icc/ifort) use –save-temp
With PGI compilers (pgcc/pgf90) use –Mnoipa=keepobj (This should be the default, but check).

5.3.3 Building codes to reduce volume of data collected

5.3.3.1 In the .apa file specify which functions to trace.

```
        -T forms_
# 2.93% 3637  bytes
        -T dftfock_
# 2.54% 910  bytes
        -T zqout_

# 2.17% 9419  bytes
        -T genr70_
# 2.14% 24939  bytes
        -T genral_
```

5.3.3.2 Prune using filters

```
#       The way traced functions are filtered can be controlled with
#       pat_report options (values used for this file are shown):
#
#       -s apa_max_count=200    No more than 200 functions are listed.
#       -s apa_min_size=800     Commented out if text size < 800 bytes.
#       -s apa_min_pct=1        Commented out if it had < 1% of samples.
#       -s apa_max_cum_pct=90   Commented out after cumulative 90%.
```

5.4 TAU Build Procedures

5.4.1 Choosing the right Makefile

For TAU there is an important directory on the DSRCs:
ERDC DSRC: $PET_HOME/pkgs/tau/craycnl/lib
Look for tau Makefiles with names like Makefile.tau-cray-papi-mpi-pdt.
export TAU_MAKEFILE=$PET_HOME/pkgs/tau/craycnl/lib/Makefile.tau-cray-papi-mpi-pdt

AFRL DSRC: $PET_HOME/pkgs/tau/x86_64/lib
export TAU_MAKEFILE=export TAU_MAKEFILE=$TAU/Makefile.tau-intel1301_mpt208-icpc-papi-ompt-mpi-pdt-openmp

NAVY DSRC:

5.4.2 Rules for AVUS

In the Makefile some changes had to be made:

> FORTRAN="tau_f90.sh"
> FORTRAN_NO_TAU="ftn"
> export TAU_OPTIONS='-optPdtF95Opts="-R free" -optVerbose'
> CC="tau_cc.sh"

And,

> FFLAGS="-G2 -h keepfiles -O2 -h noomp -f free"
> FFLAGS2="-G2 -h keepfiles -O2 -h noomp -f free"
> CFLAGS="-G2 -h keepfiles -O2 -h noomp -DLOWER_UNDERSCORE"

5.4.3 Rules for GAMESS

For building GAMESS with TAU, the install.info file generated by the use should be modified to look as follows, with the major change highlighted:

```
#!/bin/csh
#   compilation configuration for GAMESS
#   generated on garnet08
#   generated at Sat Dec 28 14:07:15 CST 2013
setenv GMS_PATH          /lustre/work1/kghosh/ABTP/app/gamess
setenv GMS_BUILD_DIR     /lustre/work1/kghosh/ABTP/app/gamess
#       machine type
setenv GMS_TARGET        cray-xt
#       FORTRAN compiler setup
#setenv GMS_FORTRAN       ftn
setenv GMS_FORTRAN       tau_f90.sh
#       mathematical library setup
setenv GMS_MATHLIB       acml
#       parallel message passing model setup
setenv GMS_DDI_COMM      mpi
setenv GMS_MPI_LIB       CrayXT
#       LIBCCHEM CPU/GPU code interface
setenv GMS_LIBCCHEM      false
```

5.4.4 Rules for HYCOM

export TAU_OPTIONS='-optPdtF95Opts="-R free" -optVerbose'

See section 5.4.4.1 below for some additional options that might be needed.
Some F90 module files (mod_dimensions mod_xc mod_za mod_pipe mod_incupd mod_floats mod_tides mod_mean mod_hycom) caused problems for the wrappers and had to be compiled with the native compiler. e.g.

```
mod_tides.o: mod_tides.F mod_xc.o common_blocks.h mod_za.o
    $(CPP) $(CPPFLAGS) $< | sed -e '/^ *$$/d' > $<.f
    $(FC) $(FCFFLAGS) -c $<.f
    mv mod_tides.F.o mod_tides.o
```

5.4.5 Keeping (Intermediate) Files for use during post-processing

With the tau compiler wrappers such as tau_cc.sh, tau_cxx.sh or tau_f90.sh users have to use the compile time option –optKeepFiles (or set the variable TAUOPTIONS="optKeepFiles –oprVerbose")

5.4.6 Building codes to reduce volume of profile data

5.4.6.1 Include list and exclude list for function

A very easy way to prune the amount of data collected is to build executables where some selected functions are traced. A file named "tau_select_functions_list" could contain

```
BEGIN_INCLUDE_LIST
GENR03
GENRAL
FORMS
DMATD
MCDAV
R30S1D
GAMESS
END_INCLUDE_LIST
```

And then a compiler option could be used as follows:

```
tau_cc.sh –optTauSelectFile="tau_select_functions_list"
or export TAU_OPTIONS=' –optTauSelectFile="tau_select_functions_list" '
```

Other examples of commands to put in an include/exclude file are:

```
#Tell tau to not profile these functions
BEGIN_EXCLUDE_LIST
void quicksort(int *, int, int)
# The next line excludes all functions beginning with "sort_" and having  arguments "int *"
void sort_#(int *)
void interchange(int *, int *)
END_EXCLUDE_LIST

#Exclude these files from profiling
BEGIN_FILE_EXCLUDE_LIST
*.so
END_FILE_EXCLUDE_LIST

BEGIN_INSTRUMENT_SECTION
# A dynamic phase will break up the profile into phase where
# each event is recorded according to the phase of the application
dynamic phase name="foo1_bar" file="foo.c" line=26 to line=27
```

```
# instrument all the outer loops in this routine
loops file="loop_test.cpp" routine="multiply"

# tracks memory allocations/deallocations as well as potential leaks
memory file="foo.f90" routine="INIT"

# tracks the size of read, write and print statements in this routine
io file="foo.f90" routine="RINB"
END_INSTRUMENT_SECTION
```

Selective instrumentation files can be created automatically from ParaProf by right clicking on a trial and selecting the Create Selective Instrumentation File menu item

6 Executing Codes to Collect Performance Data

6.1 OSS

6.1.1 Example script for submitting a batch job

```
#!/bin/bash
#PBS -q standard
#PBS -l select=43:ncpus=32:mpiprocs=32
#PBS -l walltime=04:00:00
#PBS -j oe
#PBS -V
#PBS -N hy_oss_1353
#PBS -A YourAccountNumber

export DYNINSTAPI_RT=1
export OPENSS_RAWDATA_DIR=$WORKDIR/scratch/oss/hycom_hi
rm -rf $OPENSS_RAWDATA_DIR
mkdir -p $OPENSS_RAWDATA_DIR


source ${MODULESHOME}/init/ksh
module unload PrgEnv-pgi
module load   PrgEnv-cray
#
# This step is very important or else Cray PAT libs could interfere with OSS's ability to collect data and
# you might not have a performance database at the end.
#
module unload perftools
module load /u/galarowi/privatemodules/openss
#
# Set up you app's environment input file etc. as you normally do
#
export THIS=…
export THAT=…
#
# Excute HYCOM
# Example 1. Run a PCSAMP experiment
#
```

osspcsamp "aprun -n 1353 -N 32 hycom.x"
#
or Example 2. Run a HWCSAMP experiment
#
osshwcsamp "aprun -n 1353 -N 32 " PAPI_TOT_CYC,PAPI_FP_OPS,PAPI_DP_INS

Similarly, for AVUS change the AVUS command line in the job script from
EXE=/lustre/work1/kghosh/avus/bin/avus.xt.dp
aprun -n 512 -N 16 $EXE "
to
osshwcsamp "aprun -n 512 -N 16 $EXE " PAPI_TOT_CYC,PAPI_DP_OPS,PAPI_FP_INS

6.1.2 Env Vars (experiment name is the middle part of the name of the variable)

6.1.3 Expected Output Files

If your executable is called hycom.x, OSS will create a performance database named hycom.x-pcsamp.openss, if pcsamp is the OSS experiment that was used. Other typical experiments are usertime, hwcsamp, hwc etc. If you re-run the pcsamp experiment, the next database will be named hycom.x-pcsamp-1.openss. the original database will not be overwritten.

6.1.4 Do's and don't's

Using a shared filesystem is essential.

When you submit a job to be profiled by OSS it is important to use a shared file system accessed from all nodes/ranks as follows:
export OPENSS_RAWDATA_DIR=/lustre/work1/kghosh/myopenss/mxm
rm -rf $OPENSS_RAWDATA_DIR
mkdir -p $OPENSS_RAWDATA_DIR

What if OSS numbers from HWCSAMP experiment does not match TAU or PAT?

Should you adjust the sampling rate? You could bump up the sampling rate from 100 (the default for pcsamp) to something higher, say 200 or more just as a sanity check but be aware that a higher sampling rate will lead to higher overhead and hence the numbers will be skewed. After the sanity check is done you could revert back to the default sampling rate.

6.2 PAT

6.2.1 Example batch script

```
#!/bin/ksh
#
# submit job to execute hycom.01001.pt613_reinst+apa
#
#PBS -q standard
#PBS -l select=43:ncpus=32:mpiprocs=32
#PBS -l walltime=04:00:00
#PBS -j oe
#PBS -N hy_1001_pat
#PBS -A HPCMO34140200
source ${MODULESHOME}/init/ksh
module unload PrgEnv-pgi
module load  PrgEnv-cray
module load perftools/6.1.3
```

```
module load craype-hugepages2M  # Optional
#
# Note that an instrumented binary built with pat_build must be used
#
EXE=hycom_1000.x.pat
aprun –n 1001 –N 32 $EXE     # aprun command line remains unchanged.
```

6.2.2 Env vars

Cray PAT has many environment variables that the user could specify. Some that the user could find useful are as follows.

6.2.3 Expected Output files

All initial data files are output in .xf format, with a generated file name consisting of your original program name, plus pat, plus the execution process ID number, plus the execution node number. Depending on the program run and the types of data collected, CrayPat output may consist of either a single .xf data file or a directory containing multiple .xf data files.
An example would be the directory hycom.01353_reinst+pat+24855249-1759t.
In this case the 1353 rank job has created 37 files with names like 0000NM.xf

Number of files used to store raw data

- 1 file created for program with 1 – 256 processes
- √n files created for program with 257 – n processes
- able to customize with PAT_RT_EXPFILE_MAX

6.2.4 Do's and don'ts

What to do with large number of PEs?

You can choose the rank you would like to report.
pat_report –o my_avus_000_report.txt -s='pe[000]' avus.xt.dp.instr.x+24856867-4581t

What to do when missing profile for expected functions?

Use pat_build to instrument your binary you can specify the name of the function with the –T option.
The command
pat_build -u -gmpi –T archiv_,barotp_,bigrid_,blkdat_,cnuity_,convec_,diapfl_ hycom.01001.pio
will instrument just the functions in the comma-separated list above.
The name of the executable is hycom.01001.pio and the re-built, instrumented binary will be hycom.01001.pio+pat

6.3 TAU

6.3.1 Example batch script for TAU

```
#!/bin/bash
#PBS -q background
#PBS -l select=43:ncpus=32:mpiprocs=32
#PBS -l walltime=03:00:00
#PBS -j oe
#PBS -V
#PBS -N hy_1353_tau
#PBS -A HPCMO34140200
```

```
source ${MODULESHOME}/init/ksh
module unload PrgEnv-pgi
module load   PrgEnv-cray
module unload perftools
#module load papi/5.2.0

export LD_LIBRARY_PATH=/opt/cray/papi/5.1.1/perf_events/no-cuda/lib:$LD_LIBRARY_PATH
export TAU_METRICS=TIME,PAPI_DP_OPS,PAPI_TOT_CYC,PAPI_TOT_INS
export TAU_SAMPLING=0
export TAU_VERBOSE=1
EXE=hycom.01353.tau
aprun -n 1353 -N 32 ./$EXE
```

6.3.2 Expected Output Files

TAU will generate multiple output files with performance data, one for each rank. For example, the files will be numbered profile.0.0.0 .. profile.511.0.0 if 512 ranks are used.
If, as in the job script above, TAU_METRICS is set to TIME,PAPI_DP_OPS,PAPI_TOT_CYC,PAPI_TOT_INS, There will be additional directories named MULTI__TIME, MULTI__PAPI_DP_OPS etc., each of which will have profiles files numbered profile.0.0.0, profile.1.0.0 .. profile.1352.0.0.

6.3.3 Do's and Don'ts

Use export TAU_SAMPLING=0/1 with care.
Use export TAU_THROTTLE=0/1 with care.

6.4 Code specific Suggestions

6.4.1 AVUS

6.4.2 GAMESS

6.4.3 HYCOM

6.5 Reducing Volume of Performance Data

6.5.1 MPI only

Sometimes an MPI profile is a good place to start if the executable is communication bound and there is a significant MPI activity. The MPI-only profile will also rule out if there are huge load balance issues at barriers.

6.5.2 Control Sampling Rate

There is often a direct relationship between the sampling rate and the overhead, as well as the volume of data collected and visualized. Most tools have suitably chosen defaults. You are welcome to modify the sampling rate to a higher or lower value for sanity checks and experimentation, but for running a

large code with many ranks, many functions, and functions called a millions of times, it will be necessary to choose the right sampling rate.

6.5.3 Control HWC threshold

6.5.4 Filters to ignore frequently called small functions

6.5.5 Throttle etc.

For PAT, export PAT_RT_SUMMARY=0 collects data in details.
Switching off data summarization will record detailed data with timestamps, large raw data files and significantly increased overhead.

For TAU a throttle feature (set by TAU_THROTTLE) can stop recording a function after it had been called a significant number of times.

6.6 Reducing Data Collection Overhead

6.7 Techniques For Faster I/O

Always use a parallel file system when the code is executed or the tool is running and writing to performance databases.

7 Post-processing/Visualization opportunities and utilities

7.1 Cray PAT

Recall that you had built your instrumented binary (see section ….) using
pat_build –O apa –o hycom_pat hycom
and then executed hycom as follows 9as an example):
aprun –n 1001 hycom_pat

This produces the data file my_program+pat+PID-nodet.xf, which contains basic asynchronously derived program profiling data.

pat_report is next used to process the data file.
pat_report hycom_pat+PID-nodet.xf # output to stdout
pat_report –o hycom_pat_report.txt hycom_pat+PID-nodet.xf # output to file hycom_pat-report.txt

There are three outcomes:

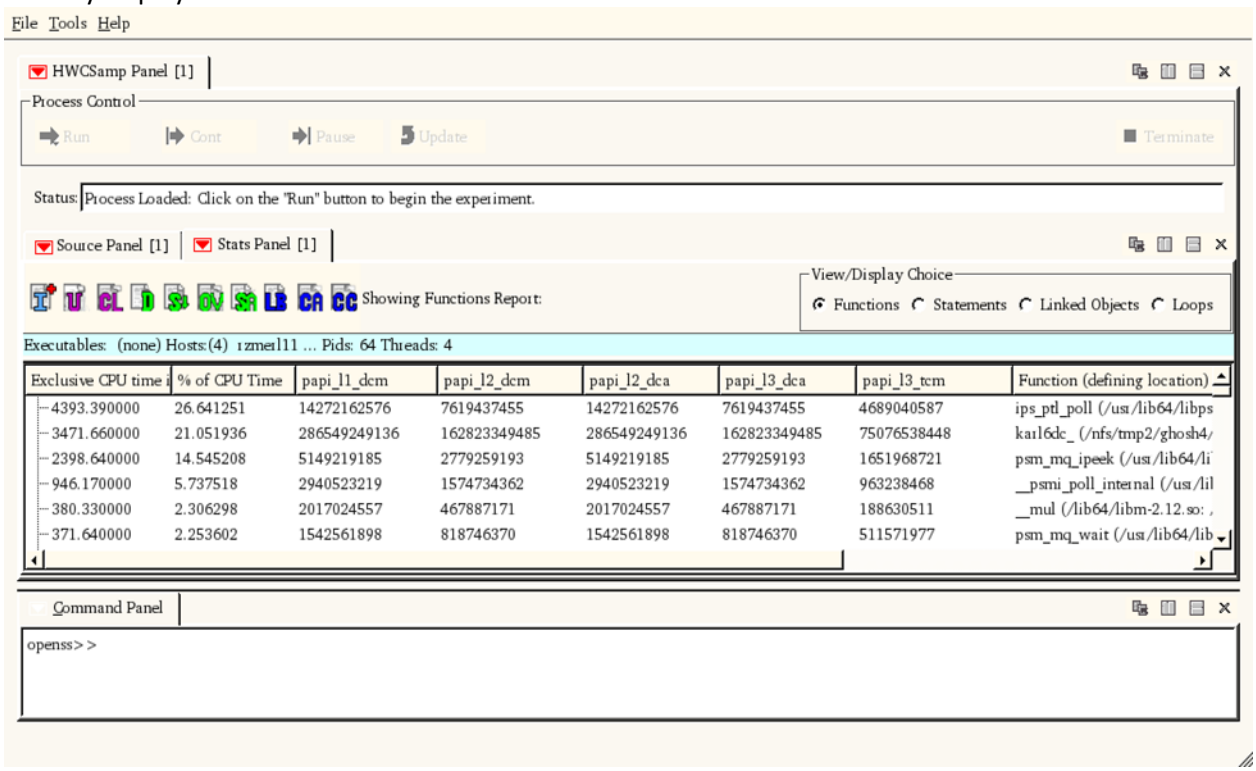
1. a sampling-based text report to stdout or to a report file
2. an .ap2 file, hycom_pat+PID-nodet.ap2, which contains both the report data and the associated mapping from addresses to functions and source line numbers, to be viewed with the powerful view tool ap2:
   app2 hycom_pat-PID-nodet.xf

3. an .apa file, hycom_pat+PID-nodet.apa, which contains the pat_build arguments recommended for further performance analysis, to be used as follows:
pat_build –O hycom_pat+PID-nodet.apa
The .apa file would information about the binary to be re-instrumented.

7.2 OSS

Recall that use of OSS does not require recompilation or relinking. During job launch one has to use helper scripts like osspcsamp, ossuertime or osshwcsamp. Here is a summary of some of the features of post-processing using OSS:

- **Launch 512 core AVUS job on GARNET**
  - module load ~galarowi/privatemodules/openssu4
  - osspcsamp "aprun –n 1001 avus"
- **Collect data from Hardware Performance Counters using osshwcsamp**
- **Display performance data**
  - module load ~galarowi/privatemodules/openssu4
  - openss avus-pcsamp.openss
  - Primary display will look like this:



- **Click on function name for Annotated source display – here we have picked L3 missies as the metric of choice**

L3 Misses	Line#	
	47	do NS=1,NUMFPC(NC)
34865687	48	NB = NABR(NC,NS)
1339704767	49	DOTPROD(2) = AMAT(6,NS,NC)*RESID(1,NB,2) + AMAT(7,NS,NC)*RESID(2,NB,2) + etc.
389097041	50	FRHS(2) = FRHS(2) - DOTPROD(2)
	51	enddo

- **Use sorting capability for displaying data (by metrics, function name etc.)**
- **Experiment comparison allows what-if analysis – use "osscompare"**

7.3 TAU

Use paraprof to postprocess TAU profile data.

If PAPI is used to collect Hardware Counter data, there will be directories with names like MULTI__PAPI_FP_OPS reflecting the PAPI preset or native counter chosen.

The paraprof viz tool should be used in the that directory. If we run a 1001-rank job, there will be 1001 files with names like profile.N.0.0, one for each rank. There could be more files if thread data is collected.

It is not necessary to view all the data which is what the bare paraprof command will do. To narrow it down, we could use

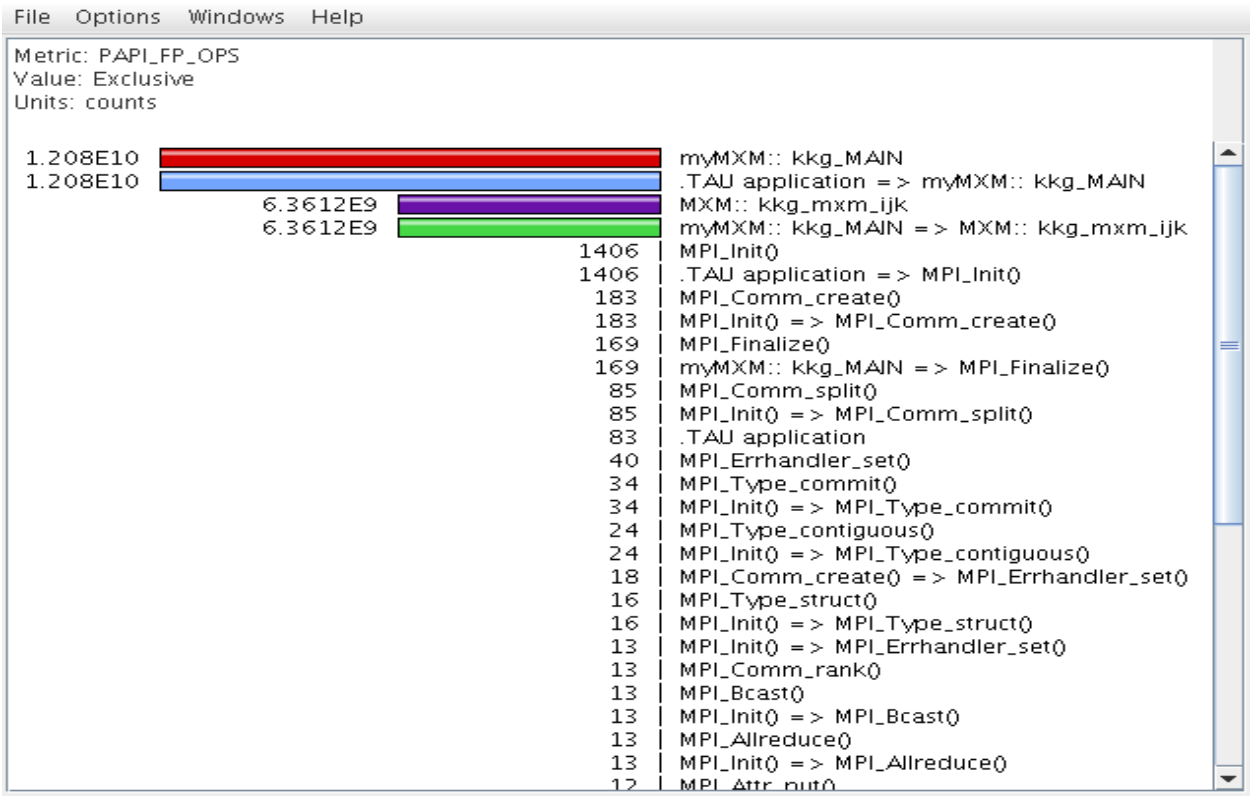
pararprof profile.0.0.0

to view rank 0 for instance.

The GUI is quite powerful, with hundreds of option which become easier to use after every attempt. Source code can displayed quite easily and intuitively. Images can be saved in jpeg format for use in presentations.

There is also a command line tool called pprof that produces searchable ASCII file often useful for processing a large volume of data. Pprof has all sorts of filters to narrow down the amount of data processed and displayed on stdout.

Below we show an image of the main display from paraprof showing the PAPI_FP_OPS count for compute intensive functions.



7.4 Dealing with Large Amounts of Profile Data

7.4.1 Select Rank

Most tools have a way to process selected ranks. You can prune the data and still obtain valuable information.

- **PAT: Examples**

pat_report –s filter_input='pe==0' –o my_report.txt gamess.x+apa+24-185t # PE# 0
pat_report –s filter_input='pe%2==0' –o my_report.txt gamess.x+apa+24-185t # EVEN PEs
pat_report –s filter_input='pe%2==1' –o my_report.txt gamess.x+apa+24-185t # ODD PEs
pat_report –s filter_input='pe<1024' –o my_report.txt gamess.x+apa+24-185t #only 1024 Pes

- **OSS: Examples**

openss –cli a.out-pcsamp.openss # command line interface will provide a prompt

- ***expview –r 0 # show me data for rank 0 only***

- **TAU: Examples**

TAU allows you to display all the data on the GUI and then select a certain rank (called a node in TAU parlance).
Or one could use the command line report-generator, pprof as follows
pprof profile.511.0.0 # show me data for rank 511
or
cd MULTI__PAPI_DP_OPS
pprof profile.511.0.0 # show me PAPI_DP_OPS data for rank 511

7.4.2 Select Performance Metric

OSS will allow users to choose a metric or a few metrics:

openss –cli a.out-pcsamp.openss ***# command line interface will provide a prompt***

- ***Expsstatus*** ***# let us see what you have stored in the database***
- ***expview –m PAPI_DP_OPS –r N*** ***# show me data for PAPI_DP_OPS for rankN***

7.4.3 Using CLI (command line interface)

8 General Strategy for Analysis of Tools

We had started the ambitious project of evaluating tools and had decided to carry on a tool-inter-comparison, looking for high quality information from all tools, making sure that tools agreed with one another. To make progress towards those goals, it was thought that we should proceed with small steps instead of one giant leap, applying a divide-and-conquer policy.
The stages that were pursued for building confidence as well as conducting sanity checks are as follows:

1. Simple kernels with known OpCount such as STREAM for calibration and sanity check
2. Mini App from LLNL e.g. UMT2013 which is complex enough but is manageably small
3. Application test-suite such as GAMESS exam01-47.inp test cases offering a wide spectrum
4. DoD relevant test-suite such as TI-14. This is the final step, using three large simulation codes, AVUS, GAMESS and HYCOM. Multiple platforms were used. TI-14 class problems were executed using high core counts (~1000).

9 Measured Results

Experiments that were/are conducted fall under these categories:

1. Compare HW counter data collected using PAPI on GARNET at ERDC. Although data was collected for AVUS, GAMESS and HYCOM, only HYCOM data is presented here.
2. Using AVUS as the application and using a TI-14 class test case, profile data on two platforms,
   a. AFRL DSRC's SPIRIT system (SGI ICE X with Intel Xeon Sandybridge @ 2.6 GHz) and
   b. ERDC DSRC's GARNET system (CRAY XE6 with AMD Opteron Interlagos @ 2.7 GHz)
   c. Compare data collected by the tools for match/mismatch
3. Compare data collected on Intel Xeon Sandybridge (AFRL's SPIRIT SGI ICE X) with that on Intel Xeon Ivybridge (NAVO's Cray XC30) – work in progress

9.1 HYCOM on GARNET (Cray XE6 w/ AMD Interlagos) Double Precision Operations using PAPI_FP_OPS

Experiment: Do the tools agree with one another?
Observation: Yes, hardware counter data (PAPI_FP_OPS) seem to match, more or less!

HYCOM Function	PAT	OSS	TAU
momtum_	13.4G	12.8 G	12.8 G
mxkprfbij	411 M	319 M	391 M
advem_fct2	2.53 G	2.85 G	2.5 G
barotp		1.08 G	1.15 G

9.2 AVUS Platform Inter-comparison Experiment (SGI ICE X w/ Sandybridge vs. Cray XE6 w/ Interlagos) Measuring PAPI_DP_OPS

AVUS HW Counter (PAPI_DP_OPS)	Open Speedshop		TAU	
Function	SPIRIT SGI ICE X Intel Sandybridge	GARNET CrayXE6 AMD Interlagos	SPIRIT SGI ICE X Intel Sandybridge	GARNET Cray XE6 AMD Interlagos
walldst_	3575 G	3126 G	3133 G	3129 G
karl6sc	1.30 G	1.098 G	1.51 G	1.125 G
ucm6_	.470 G	.455 G	0.597 G	0.523 G

10 Which Tools To Use?

- A first-time user does not care if the tools uses sampling or tracing or binary instrumentation. They do care about these features:
  - No special re-compilation
  - No special re-linking
  - No tricky instrumentation
  - Fast ramp-up time - use their batch scripts without adding too many complications
  - Easily launch their jobs and collect performance data
  - Gather, without gymnastics, basic performance information about time spent in
    - MPI

        - Compute-intensive functions
        - Statements
        - Loops
        - I/O
    - Easy to use GUI for post-processing / viewing of collected data
    - Insight – how can this code be sped up?
- More advanced users care about ease of use too but have more flexibility and needs:
    - Re-compilation, re-linking, instrumentation acceptable if tool provides useful details about
        - Functions, loops, statements, libraries
        - Annotated source code
        - Parallel regions, OpenMP, GPU etc
        - Events timeline
        - Insight
        - Feedback to tool and iterative tuning
    - Caller-callee relationship/trees
    - Good scaling with large core counts
    - Choice of remote / local post-processing
    - Hardware Performance Counters for exposing bottlenecks
    - Support and installation
    - Sampling + tracing in a single framework
    - Transparent instrumentation (pre-loading and binary)
    - Insight – bottlenecks? BW, latency, TLB miss, stalls? how can this code be sped up?
- Insight is currently the hardest job for a tool.

11 Future Work

- Threads: Study OpenMP performance using TAU/PAT or (scoreP , HPCToolkit)
- Vectorization: analyze using native HW counters
- Measuring Mem BW for Intel processors
    - Known tough problem for Intel processor
    - BW easily measured for IBM BG/Q or Opterons
- What about memory latency and latency hiding information
    - New memory tools (LLNL/CASC? Google gooda?)
- Using newer technologies
    - Integration of new features into the OS to be used by new tools
    - PEBS, RAPL, LLNL's Dr. Roundtree and his power measurement tools
- Predict GPU suitability of codes using performance counter data?

12 Acknowledgements

- DoD HPCMP
    - Dr. Roy Campbell
    - Dr. William Ward
- ERDC CS&E HPC Performance Specialists
    - Dr. Tom Oppe
    - Laura Brown
    - Carrie Leach

- Tool Developers
    - Sameer Shende (Paratools/University of Oregon)
    - Jim Galarowicz, Don Maghrak (Krell Institute)
    - Heidi Poxon (Cray Inc.)

13 Documentation

13.1 Cray PAT

Using Cray Performance Measurement and Analysis Tools, http://docs.cray.com/books/S-2376-60, or http://docs.cray.com/books/S-2376-60//S-2376-60.pdf , Dated Sep 2012. Explains how to use CrayPat, Cray Apprentice2, and Reveal to capture, evaluate, and understand program performance on Cray XE and Cray XK systems. It is a hundred-page document packed with useful information.

Also see the tutorial "Performance Measurement andAnalysis Tools for the Cray XK System", Heidi Poxon, Cray Inc. at https://www.olcf.ornl.gov/wp-content/uploads/2013/01/performance_tools2.pdf

Alternatively, you may create your Cray PAT documentation using the pat_help command:

```
pat_help all . > all_pat_help.txt
pat_help report all . > all_report_help.txt
```

The man page for pat is also a source of useful information (man pat | col –b > craypat_man.txt).

13.2 OSS

There is a Quick Start Guide and a detailed User Manual available online:
http://www.openspeedshop.org/wp/wp-content/uploads/2013/12/OSSQuickStartGuide2013revised.pdf

http://www.openspeedshop.org/wp/wpcontent/uploads/2014/02/OpenSpeedShop_2.1_User_Manual.pdf

13.3 TAU

TAU Tutorial is here: **http://tau.uoregon.edu/tau.ppt**
More about TAU can be found on the About Page, or browse the Online documentation on the Documents Page.

13.4 Allinea MAP

Allinea DDT and MAP User Guide
http://content.allinea.com/downloads/userguide.pdf